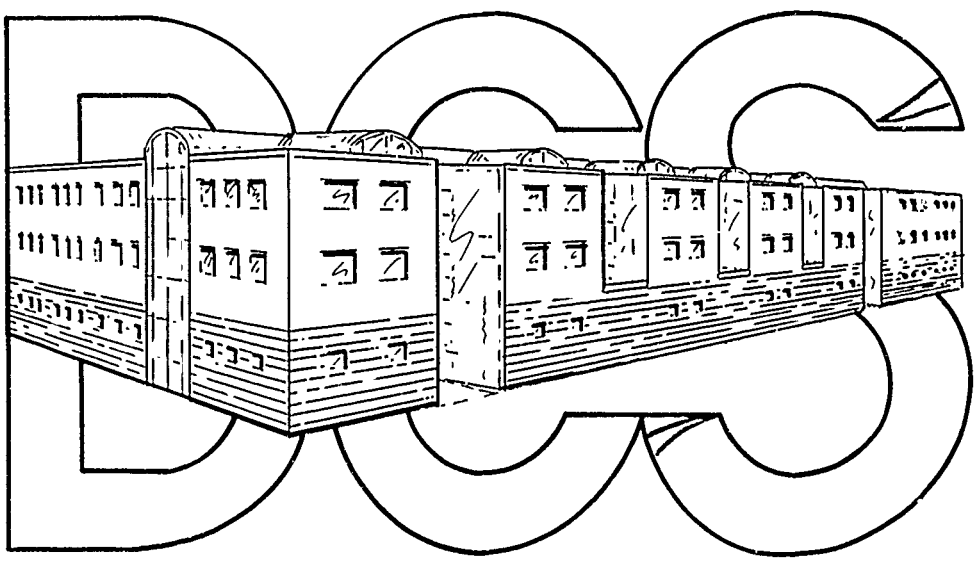
DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN



THE NEW ADDITION

REPORT NO. UIUCDCS-R-90-1626 UILU-ENG-90-1765

TOWARDS INTELLIGENT FINITE ELEMENT ANALYSIS

by

Alejandro G. Blanco

September 1990

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UIUCDCS-R-90-1626	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Dept. of Computer Science University of Illinois 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1304 W. Springfield Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy Street Arlington, VA 22217-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-86-K-0309
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Towards Intelligent Finite Element Analysis

12. PERSONAL AUTHOR(S)
Alejandro G. Blanco

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	September 1990	56

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
05		10	Finite Element Methods, Machine Learning, Explanation-Based Learning, Intelligent Numerical Methods.

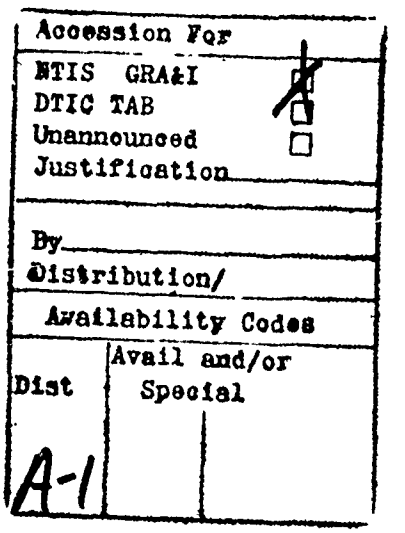
19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The aim of this research is the use of various artificial intelligence (AI) techniques to improve the performance of programs performing finite element analysis, a method that numerically solves differential equations that govern a physical situation. Typically, a finite element program requires extensive analysis and frequent interruption by a scientist, which reduces how much the scientist can learn about the phenomenon being studied by consuming the scientist's resources and time in the drudgery of mathematically conditioning the input to the finite element program. This work usually has no relation to the matter being studied. This thesis investigates whether AI can be used to reduce the time spent doing this type of work.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Alan Meyrowitz	22b. TELEPHONE (Include Area Code) (202) 696-4302 22c. OFFICE SYMBOL ONR

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

TOWARDS INTELLIGENT FINITE ELEMENT ANALYSIS



BY

ALEJANDRO G. BLANCO

B.S., University of Miami, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

DEDICATION

A Mami y Papi.
¡Que Dios los bendiga!

ACKNOWLEDGEMENTS

The author wishes to acknowledge the enormous amount of input and help received from his advisor, Professor Gerald DeJong. Others who have provided insightful comments include Michael Barbehenn, Scott Bennett, Steve Chien, Melinda Gervasio, Jon Gratch, and Dan Oblinger. The Cure is credited for inspiration or lack thereof.

The research reported in this paper was supported by the Office of Naval Research under grant number N0001486-K-0309. The author was supported by an AT&T Fellowship.

TABLE OF CONTENTS

1. INTRODUCTION

The aim of this research is the use of various artificial intelligence (AI) techniques to improve the performance of programs performing finite element analysis, a method that numerically solves differential equations that govern a physical situation. Typically, a finite element program requires extensive analysis and frequent interruption by a scientist, which reduces how much the scientist can learn about the phenomenon being studied by consuming the scientist's resources and time in the drudgery of mathematically conditioning the input to the finite element program. This work usually has no relation to the matter being studied. This thesis investigates whether AI can be used to reduce the time spent doing this type of work.

When using finite element analysis, a scientist generally follows three steps. In the first step, the scientist will choose a numerical experiment to run. The second step involves running the experiment by inputting specifications to a finite element program. The third step involves analyzing the results of this program and deciding if any additional experiments should be run. When analyzing the finite element results, the scientist must be sure that the results are plausible to preclude round-off errors and ill-conditioning. This three-step process continues until a sufficient understanding of the problem area has been attained.

The approach of this thesis is to concentrate on the second step. Modern finite element analysis involves the calculation of many numbers. For this reason, a good deal of effort is appropriately invested in making finite element programs run as efficiently as possible. One goal of this thesis is to explore the use of AI to allow finite element programs to run more efficiently. A scientist must construct numerical experiments and adopt corresponding techniques to ensure the viability of the

results. An intuition behind this thesis is that common sense in the form of qualitative reasoning [Forbus84] can capture the knowledge a scientist uses in this process. Therefore, the main goal of this thesis is to design a domain-independent finite element program that uses a domain theory about the phenomenon being studied to intelligently solve the problems presented to it.

Solving finite element analysis problems is inherently repetitive, both within and across runs. Therefore, a properly designed system might be able to learn from experience to increase its efficiency. However, identifying what to learn, how to learn it, and how to use it are difficult tasks. Explanation-based learning (EBL) [DeJong86] is a knowledge-intensive form of machine learning. Performing EBL with the domain theory described above is a candidate for learning from experience to increase efficiency.

The rest of this introduction consists of three sections supplying background in finite element analysis, explanation-based learning, and qualitative reasoning. Chapter 2 of this thesis delineates the methodologies taken in conducting this thesis, including the description of a novel finite element object-oriented solver named FEOS. Chapters 3, 4, and 5 describe various approaches to reasoning and learning to improve the solver of Chapter 2. Chapter 6 then describes future work and some conclusions.

1.1 Finite Element Analysis

1.1.1 Definition

Finite element analysis is the use of numerical techniques to approximately solve a set of differential equations governing a physical phenomenon. Examples of physical phenomena for which finite element analysis is used include material deformation, heat transfer, electromagnetic

field dissipation, and fluid flow. The area where the phenomenon occurs is divided into *elements* of finite size, whose borders are defined by *nodes*. The number of nodes that define an element is known as the *degree of freedom* of that element. The set of all elements in a problem is referred to as the *mesh*. A mesh may be comprised of elements of different shapes, sizes, and degrees of freedom. Figure 1 summarizes these definitions.

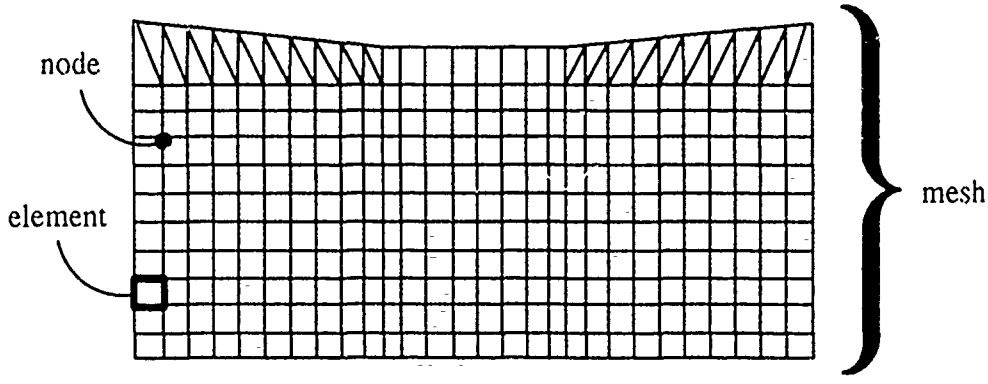


Figure 1. A Sample Mesh

The goal of finite element analysis is to find the values of the unknown parameters at the nodes and, therefore, have the whole solution by interpolation across each element. Local linear algebraic equations defining the relationship between the parameters at the different nodes within an element are derived from the differential equations that govern the physical phenomenon by assuming small parameter variations across the element. The small variation assumption requires small elements and this necessitates the large number of elements that are typical of finite element problems. These equations can be viewed as *local constraints* on the parameters. Since nodes are shared by neighboring elements, and the values for the parameters at a node must be the same for all elements that share the node, there are *compatibility constraints* on the parameters to ensure

consistent solutions. Additional constraints are supplied by *boundary conditions*, which are parameter values at particular nodes specified as part of the problem. The solution at each node is found by simultaneously solving the set of local equations after application of the boundary conditions and ensuring that the compatibility constraints are satisfied. This is known as *solving the mesh*. Traditional finite element programs solve a mesh by assembling the local equations into a *stiffness matrix*, K, replacing any values known by boundary conditions, and then solving the equations using highly optimized matrix manipulation techniques. An example showing the stiffness matrix used in traditional methods is included in the following section. Finite element programs spend most of their run time in solving these equations.

1.1.2 An example problem class

To facilitate analysis, a simple one–dimensional problem class is used as an example. The problem is a beam under loads. Figure 2 shows a three–element single force example and the node

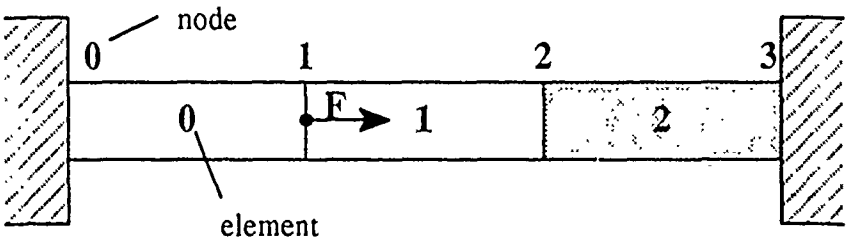


Figure 2. Three–Element Example and Numbering Conventions

and element numbering conventions. The unknowns are

(1) the stress, or the internal force or tension, for each element,

(2) the strain, or the change in length due to compression or expansion, for each element,

(3) the external force at each node,

(4) and displacement at each node.

Locally, the stress, f, positive for expansion by convention, and the strain, which is the difference between the right displacement, d_R, and left displacement, d_L, both measured positive to the right, are related by

$$f = k(d_R - d_L), \tag{1}$$

where

$$k = AE/L, \tag{2}$$

A = the cross-sectional area,

E = the modulus of elasticity, and

L = the original length.

Figure 3 summarizes the variable naming conventions. The compatibility constraints are that neighboring elements, i and $i+1$, must agree on the nodal displacement they share, and their stresses must equal the external forces at the node, $i+1$, in this case. These can be written as

$$d_R[i] = d_L[i+1], \tag{3}$$

and

$$F[i+1] = f[i] - f[i+1], \tag{4}$$

respectively. In traditional finite element analysis, the notation would be

$$x_i = d_R[i-1] = d_L[i], \tag{5}$$

and

$$F_i = F[i+1]. \tag{6}$$

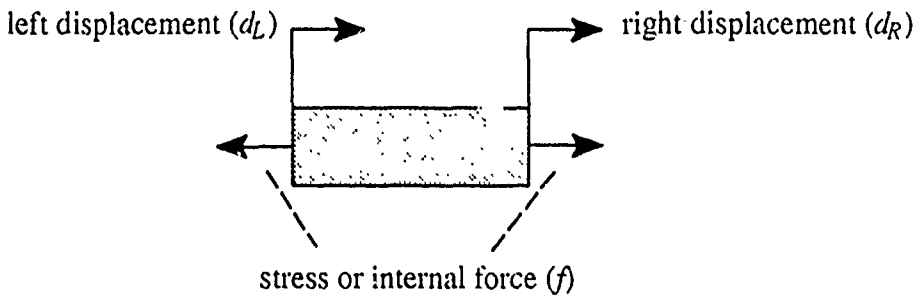


Figure 3. Variable Naming Conventions

This results in

$$\bar{F} = \begin{bmatrix} F_1 \\ \vdots \\ F_n \end{bmatrix} = K\bar{x} = \begin{bmatrix} k_1 & -k_1 & 0 & \cdots & 0 \\ -k_1 & k_1+k_2 & -k_2 & \ddots & \vdots \\ 0 & -k_2 & k_2+k_3 & \ddots & 0 \\ \vdots & \ddots & -k_3 & \ddots & -k_{n-1} \\ 0 & \cdots & 0 & -k_{n-1}+k_n & -k_n \\ & & & -k_n & k_n \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

as the set of equations to be solved for an n–element problem, where K is the stiffness matrix. The boundary conditions would then be inserted and the set of equations would be solved for the unknown F_i's and x_i's.

1.2 Explanation–Based Learning (EBL)

Explanation–based learning [DeJong86] is learning with background knowledge. When an intelligent being encounters an example of something it wishes to learn, it references its memory in an attempt to understand the example. If the being is able to build an explanation for the example in terms of its background knowledge of the world, then the being can learn from the example by

remembering the conditions under which the explanation holds and the resulting information. This can then be summarized by a rule that has *preconditions*, or *antecedents*, and *consequents*. In this way, the explanation does not have to be regenerated when the learned information is desired. Additionally, the preconditions and consequent can be generalized so that the new rule can apply to more than just the example seen.

The task is more rigorously stated in Table 1. The explanation generation task is one of sim-

---

Table 1. The Explanation–Based Learning Task [Mitchell86]

Given:

- *Target Concept*
- *Training Example*
- *Domain Theory*: a set of rules and facts used to explain the training example
- *Operationality Criterion*: a specification of the form of the learned rule

Determine:

- A generalization of the training example that is a target concept description and satisfies the operationality criterion

---

ply proving the example is correct using memory contents expressed as facts and rules. Several methods for learning and generalization are available [Mitchell86], [Mooney86], but the EGGS method [Mooney86] is preferred because of its efficiency and simplicity. The basic idea is to search until an explanation has been found. The top of the explanation forms the consequent and the bottom

forms the antecedents. Generalization is accomplished by removing any variable bindings to the specific example, thus retaining only those necessary for the explanation to remain valid.

A classic example, taken from Mitchell et al. [Mitchell86], is that of recognizing a cup. Using the domain theory in Table 2,[1] one can explain the training example in Table 3 with the explana-

Table 2. A Partial Domain Theory[1]

liftable(X) ∧ stable(X) ∧ open-vessel(X) ⇒ cup(X)
is(X,light) ∧ part-of(X,Y) ∧ is-a(Y,handle) ⇒ liftable(X)
part-of(X,Y) ∧ is-a(Y,bottom) ∧ is(Y,flat) ⇒ stable(X)
part-of(X,Y) ∧ is-a(Y,concavity) ∧ is(Y,upward) ⇒ open-vessel(X)

• • •

Table 3. A Cup Training Example

cup(object1)	is(object1,light)	part-of(object1,handle1)
part-of(object1,concavity1)	part-of(object1,bottom1)	is-a(handle1,handle)
is-a(bottom1,bottom)	is-a(concavity1,concavity)	is(bottom1,flat)
is(concavity1,upward)		

• • •

tion in Figure 4. The learned rule from this example is

is(X,light) ∧ part-of(X,Y) ∧ is-a(Y,handle) ∧ part-of(X,Y) ∧ is-a(Y,bottom)

∧ is(Y,flat) ∧ part-of(X,Y) ∧ is-a(Y,concavity) ∧ is(Y,upward) ⇒ cup(X).

[1] ∧ is the symbol for conjunction, or logical AND, and ⇒ is the symbol for logical implication

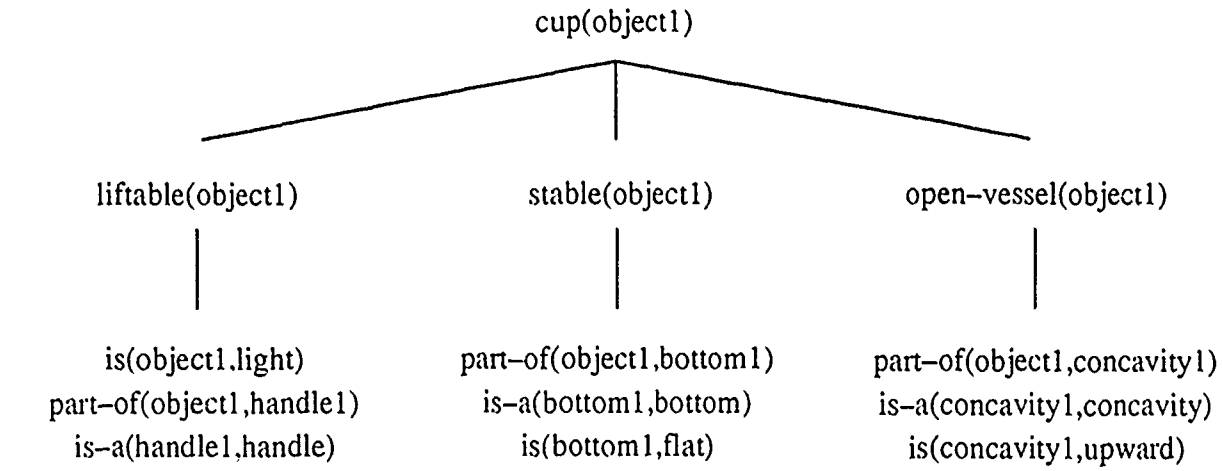


Figure 4. Explanation For the Cup Example of Table 3

The concept of an operationality criterion is one that has been difficult to define. However, as can be seen from the example, it is necessitated by the fact that a nonoperational definition of the target concept is already known. This leads to the conclusion that what is to be learned is essentially already known. Therefore, an operationality criterion defines the utility of EBL in that the goal is to *operationalize* a nonoperational concept description. Although this may seem of little value, the fact is that with a large amount of information, the concept may never be discovered without the example. Explanation of the example is a highly constrained search through this information because many hypothetical considerations are not encountered.

In order to use EBL to improve finite elements, the task must be rigorously stated as suggested by Mitchell et al. [Mitchell86]. The inputs should be defined as

- *Target Concept*: good decisions at choice points in a finite element program,
- *Training Example*: examples from previously solved problems,

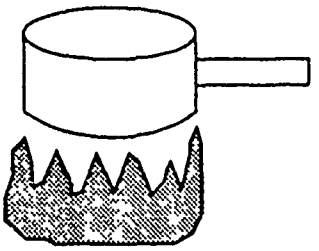
- *Domain Theory*: a domain theory that can explain decisions must be used. Qualitative reasoning, described in the next section, is a candidate for this domain theory.
- *Operationality Criterion*: preconditions that can be recognized at the decision points.

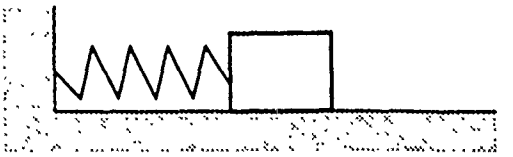
1.3 Qualitative Reasoning

Qualitative reasoning [Forbus84] is reasoning about situations nonnumerically. Through reasoning about confluences between parameters, many interesting conclusions, such as possible future states or causes for state changes, can be determined. Some modified examples, taken from Forbus [Forbus84], of questions that can be answered with qualitative reasoning are shown in Figure 5. As shown in these examples, qualitative reasoning can be used to explain as well as predict physical situations.

Confluences between parameters are represented as qualitative proportionalities, referred to as "q-props," or inverse qualitative proportionalities. Whenever there is a nonconflicting set of q-props and inverse q-props to a parameter, X, it is known to be increasing, decreasing or constant. A nonconflicting set occurs when all the nonconstant parameters that are q-prop to X are all increasing or all decreasing, and similarly for nonconstant parameters that are inverse q-prop to X, and that if the nonconstant q-prop parameters are increasing, then the nonconstant inverse q-prop parameters are decreasing or vice versa. Table 4 shows a domain theory that encodes the nonconflicting set of q-props (Q+) and inverse q-props (Q-) concept and supports the conclusions made in the first example of Figure 5. An explanation structure for question Q2 of Figure 5 is shown in Figure 6.

Many of the available qualitative reasoning techniques have not been used in this research. The usefulness of qualitative reasoning for finite element analysis is in its ability to codify the type





Q1: What might happen?

A: The water inside might boil, and if the pot is sealed it might blow up.

Q2: Why might it blow up?

A: A sealed pot would fix the volume and mass, and pressure would increase solely with temperature. If the pressure goes above a certain value, the pot will blow up.

Q3: What happens if the block is released?

A: Assuming the spring doesn't collapse, the block will oscillate back and forth, and if there is friction, it will eventually stop.

Q4: Why would friction cause it to stop?

A: Friction produces a force in a direction opposite to that of motion. A force will cause an acceleration in its direction, and thus the frictional force will cause a deceleration. Since the amount the block moves is directly proportional to its velocity at the beginning of each oscillation and the deceleration is reducing this velocity, the block will eventually stop.

Figure 5. Some Questions Qualitative Reasoning Can Answer

of knowledge used by a scientist when designing finite element programs. This is done through the definition of qualitative domain theories that can be used with EBL to explain and learn about decisions made in solving finite element problems.

Table 4. A Partial Domain Theory and Base Facts for Question Q2

Domain Theory:

Q+(temperature,pressure)
Q+(heat,temperature)
Q–(volume,pressure)
Q+(mass,pressure)

value–for(X,Y,Z) ∧
(∀ W, Q+(W,Y) value–for(V,W,Z) ∧ (increasing(V) ∨ constant(V))) ∧
(∀ W, Q–(W,Y) value–for(V,W,Z) ∧ (decreasing(V) ∨ constant(V)))
⇒ increasing(X)

value–for(X,Y,Z) ∧
(∀ W, Q+(W,Y) value–for(V,W,Z) ∧ (decreasing(V) ∨ constant(V))) ∧
(∀ W, Q+(W,Y) value–for(V,W,Z) ∧ (increasing(V) ∨ constant(V)))
⇒ decreasing(X)

X < Y ∧ increasing(X) ⇒ eventually–greater(X,Y)

being–heated(X) ∧ value–for(Y,heat,X) ⇒ increasing(Y)

sealed(X) ∧ contains(X,Y) ∧ value–for(Z,volume,Y) ⇒ constant(Z)

sealed(X) ∧ contains(X,Y) ∧ value–for(Z,mass,Y) ⇒ constant(Z)

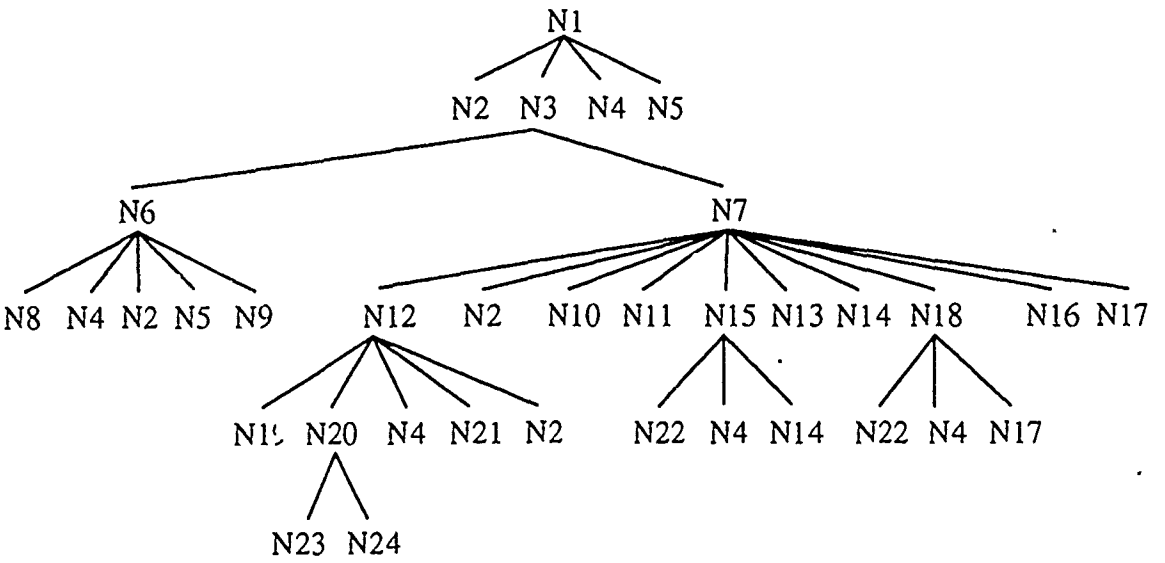
value–for(X,temperature,Y) ∧ increasing(X) ∧
noninsulating(X) ∧ contains(X,Y) ∧ value–for(Z,temperature,Y) ⇒ increasing(Z)

value–for(X,pressure,Y) ∧ contains(Y,X) ∧
bursting–pressure(Y,P) ∧ eventually–greater(X,P) ⇒ eventually–blows–up(Y)

intact(X) ∧ contains(X,Y) ∧
value–for(Z,pressure,Y) ∧ bursting–pressure(Y,PB) ∧ value(Z,P) ⇒ P < PB

Base Facts:

value–for(c–pressure,pressure,contents)
contains(pot,contents)
being–heated(pot)
noninsulating(pot)
bursting–pressure(pot,pb)
intact(pot)
value(c–pressure,p)
sealed(pot)
value–for(c–temperature,temperature,contents)
value–for(c–volume,volume,contents)
value–for(p–temperature,temperature,pot)
value–for(c–mass,mass,contents)



Nodes:

N1:	eventually–blows–up(pot)	N13:	Q+(mass,pressure)
N2:	value–for(c–pressure,pressure,contents)	N14:	value–for(c–mass,mass,contents)
N3:	eventually–greater(c–pressure,pb)	N15:	constant(c–mass)
N4:	contains(pot,contents)	N16:	Q–(volume,pressure)
N5:	bursting–pressure(pot,pb)	N17:	value–for(c–volume,volume,contents)
N6:	c–pressure < pb	N18:	constant(c–volume)
N7:	increasing(c–pressure)	N19:	value–for(p–temperature,temperature,pot)
N8:	intact(pot)	N20:	increasing(p–temperature)
N9:	value(c–pressure,p)	N21:	noninsulating(pot)
N10:	Q+(temperature,pressure)	N22:	sealed(pot)
N11:	value–for(c–temperature,temperature, contents)	N23:	being–heated(pot)
		N24:	value–for(p–heat,heat,pot)
N12:	increasing(c–temperature)		

Figure 6. Explanation for Question Q2 of Figure 5

2. METHODOLOGIES

This section describes a novel finite element solver and the methodologies used in trying to improve its performance via application of AI techniques. These techniques have all been tested for the example problem class described in Section 1.1.2. More complex problem classes might be better targets for the application of AI techniques because there are more areas where complexity can be reduced. Additionally, the techniques found useful in this domain may not be scalable to more complex domains. However, the simple problem class allows discussion of many introductory issues and allows implementation of more techniques. The issue of scalability is addressed for each of the candidate techniques described.

2.1 Object–Oriented Programming

Object–oriented programming [LaLonde90] involves defining data items as *objects* which are instances of *object types* and any actions to be performed on these data items as *operations* associated with the object type. Once the object types and operations have been defined, a task can be delineated in terms of operations to be performed on objects. In top–down programming, once lower level procedures and functions have been written and debugged, they are called to perform tasks without regard to how the tasks are performed. Similarly, in object–oriented programming, once the object types and operations have been written and debugged, they are called without thought of how they perform the desired tasks.

2.2 The Finite Element Object–Oriented Solver (FEOS)

In order to facilitate the study of various approaches to learning for finite element analysis, a novel finite element solver, named FEOS, has been designed. FEOS is an object–oriented solver

that represents each element as an object and has operations that contain the local constraints. FEOS has been implemented in the Flavors System of Lucid Common Lisp on an IBM RT,[2] and the code has been included in APPENDIX A. The elements enforce the compatibility constraints by communicating with neighboring elements. Essentially, an element propagates shared parameter values to its neighbors whenever it changes any of its parameters, while maintaining its local constraints. In this way, when no more messages are being sent, all compatibility constraints have been met and the solution has been converged upon. The main activity of an element upon the receipt of values from its neighbors is summarized by the following pseudo code:

```
if values are inconsistent with compatibility constraints
   compensate for compatibility constraints
         with decision procedure Φ
   re-establish local constraints
         with decision procedure Γ
   send message(s) to neighbor(s)
endif
```

The element may have several alternate ways of accomplishing the compensation and reestablishment steps. For example, suppose element A receives a message that a neighbor, B, has a value, x_B, for the displacement at the bordering node, and that A has a value, x_A, for the displacement. If $x_A \neq x_B$, A can change its value to x_B and compensate by changing its stress to agree with the new strain. It could, on the other hand, set its value for the displacement to any value between x_A and x_B and again

[2] RT is a trademark of International Business Machines Corporation

compensate by changing its stress. It could, as well, for both of the above alternatives, absorb the change in the displacement only partially with the stress and partially with a change in its other displacement. These choices are all seemingly arbitrary at the element level, but certainly will affect the speed of convergence. This example shows the necessity of the decision procedures, Φ and Γ. The decisions Φ and Γ must make are

(1) whether to meet, undercompensate, or overcompensate for an incompatibility, i.e. given a parameter, A, with a value x, and the receipt of a message with a value of $y > x$ for A, should the new value for A be y, $x < z < y$, or $z > y$, and

(2) which parameters to change given an incompatibility.

Experimentation in terms of efficiency and convergence results in heuristic default definitions for these decisions that have been built into FEOS. The defaults are that Φ specifies setting the contested parameter, A, to the average of its value and the value received in the message, $(x+y)/2$, and Γ specifies changing the stress first, if possible, and as little else as possible to compensate. There are no analytical claims made about convergence of FEOS on a solution, but empirically FEOS converges on the solution for 98% of all examples run.

A system that is to use AI techniques must be able to reason about

(1) a single element,

(2) all elements, or

(3) a set of elements.

In traditional finite element techniques, the elemental equations are assembled into a matrix that must then be solved. Therefore, there is no way to insert additional knowledge into an element. The fact that FEOS is object–oriented and the way it is organized facilitate the application of AI techniques to finite element analysis. With FEOS the types of reasoning described above can each be addressed by

(1) reserving an area in the element object type that is to contain element–specific knowledge to be used by the operations,

(2) reserving a global area to be used by the operations, and

(3) either of the two above methods with a membership function to determine which elements use which parts of the knowledge if a global area is used.

Additionally, elements can be reasoned about in a natural fashion because the division of the elements into objects encapsulates all elemental information. Specifically, all parameter values and convergence steps can be analyzed at the individual elements.

With FEOS, the task of improving finite element analysis becomes one of improving FEOS. The remainder of this thesis describes different approaches to inserting intelligence into FEOS in an attempt to improve its performance.

2.3 Improving Convergence

Section 2.2 identified several decisions for which FEOS has heuristics. One approach to improving FEOS is to improve its convergence behavior by learning justifications for these decisions. One way to attack this task is to learn additional constraints for the elements. The goal in learning constraints is to add global constraints to the local constraints already known at the elemental level.

These constraints can then be used to restrict the possible decisions. For example. consider a constraint that requires that an elemental parameter, A, with a value x_A, greater than or equal to some number, x_L, and a message with a value x_B for A arrives. If $x_B < x_L$, then A cannot be set to x_B. Additionally, it is known that x_B is an incorrect value for A.

A quantity lattice [Simmons86] can take as input the equations and boundary conditions of a problem and determine many constraints, such as bounds on parameters and inequalities and equalities that exist between different parameters. However, it is possible to establish an arbitrary number of constraints between arbitrary parameters and expressions involving parameters. Therefore, it is necessary to determine which additional constraints would be useful. Two specific types of constraints have been identified. Studies of their utility are presented in subsequent sections. The basic premise is that qualitative knowledge can be used to determine which constraints to learn.

2.4 Meshing Issues

Meshing issues come down to the attempt to resolve a tradeoff between minimization of mesh density and accuracy, where mesh density is the number of elements per mesh area. The idea is that less calculation is necessary to solve a mesh with fewer elements, while accuracy goes down with a coarser mesh, because the assumption that parameters have small variations across an element used to derive the local constraints is less valid across larger elements.

Resolution of this tradeoff can be approached with intelligent initial meshing. Although there is no inherent advantage for FEOS in terms of solving a problem once a mesh has been determined, the determination of which elemental borders should be removed is facilitated by the object–oriented approach used in FEOS.

A second approach to resolving the meshing tradeoff is the use of a dynamic mesh that allows elements to fuse and split during solution of the equations. This approach is not implementable with traditional finite element solution techniques because any change in the mesh during solution would require the assembly of new matrices and any work done to that point would be meaningless. FEOS can easily fuse or split elements by simply retaining all intermediate parameter values and interpolating values for any new nodes. Convergence can then continue from this point. Fuse/split decisions can be made and learned about. Although this thesis does not contain any results on dynamic mesh proposals, it appears that the use of a qualitative domain theory that can explain how fuse/split decisions can affect gradients is a promising avenue for future research.

3. DERIVING APPLICABLE EQUATIONS

One approach to learning a specialized constraint is to learn a ratio, θ, between two elemental parameters, X and Y, adding the constraint $X = \theta Y$. It must be known that X and Y have a linear relationship, but this type of relation is very common in finite element analysis because of the formulation of the problem into a set of linear equations. For the example problem class, boundary conditions specify either the displacement or the external force at each node and the other is an unknown. Therefore, for every element, X is the unknown for the left node and Y is the unknown for the right node. A domain theory, reproduced in APPENDIX B, has been developed that explains θ for an element by explaining why X and Y take on the values that they do. The domain theory is composed of rules of the following form:

```
value(Element,Z,z) ⇐
    p1(Element)              ; preconditions
    ...
    pn(Element)
    f1(x1,Element)           ; subexpression
    ...
    fm(xm,Element)
    z = f(x1,...,xm)
```

Rules of this form say that if conditions p_1 through p_n hold for *Element*, then the elemental parameter $z = f(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are calculated via $f_1, \ldots f_m$. Explaining a ratio, therefore, is equivalent to deriving an expression for it, namely, $f(x_1, \ldots, x_n)$. This expression is in terms of subexpressions, $x_1, \ldots, x_m$, that must also be derived in terms of subexpressions, until $x_1, \ldots, x_n$ are all elemental parameters. The number of subexpressions and preconditions for z is $(n+m)$, usually a small constant. However, the number of levels of subexpressions will grow with the number of elements and since the solution must be searched for, the search is exponential in the number of elements. This implies that although the method may work for small examples, it will not scale up to more realistic examples because an exponential space must be searched.

Since the search is through a large space, this task would normally be a good candidate for EBL which can help guide the search through the use of the example being explained. However, in this case, the example is the ratio from a solved problem in the form *ratio(elementX,x)*, where x is a number. The numerical answer cannot be tested against alternatives at a node in the search space since the alternatives represent different expressions and their numerical values cannot be determined until the expression is known. The expression is known only after the subtree at the alternative has been searched. Therefore, the utility of searching an alternative cannot be precomputed and thus no search can be saved.

A system that explores these ideas, and on which many of the results described are based has been designed and implemented. It consists of an EBL Lisp program that explains examples and learns from them, and the domain theory in APPENDIX B. The problem of complex rules becomes

very evident from a direct application of EBL to this domain theory. A three–element program produces rules with 80 antecedents. However, it may be surmised that many of these antecedents are superfluous due to subsumption by other antecedents in the rule. The application of truth–preserving subsumption–removal transforms to the aforementioned rule results in only 19 antecedents remaining. The form of this rule, however, shows that there are conclusions drawn from the state of each element and thus the complexity of the rule is in terms of the number of elements.

4. QUALITATIVE STATES

Qualitative reasoning allows one to conclude that an object is in a certain state. Once the state of an object is known, certain equalities and inequalities can be known to constrain the object. One approach to learning specialized constraints is to detect when an element is in a qualitative state and use the implied constraints to restrict choices.

For the example problem class, an element can be provably in compression or expansion. Proving an element is in compression or expansion involves external forces and the fact that an element is a member of a bar that is eventually fixed on one side. For example, in Figure 2, elements 1 and 2 are provably under compression and element 0 is provably under expansion. An explanation for element 0 being under expansion is shown in Figure 7. A domain theory that can be used to determine the compression/expression states of elements is included in APPENDIX C.

If an element is known to be in compression or expansion, then the sign of the stress is known. Therefore, whenever an incompatibility is encountered by an element, it can restrict compensating parameter changes to only those which will not cause the sign of the stress to be the opposite of what is known to be the final sign. More specifically, whenever the default Φ and Γ propose the alteration of the stress, the state of the element after the proposed alteration is checked against the known final state, and alternative compensations are performed if these states conflict. Alternatives are the changing of parameters other than the stress. Three versions of FEOS exist. They are

- FEOS, which has the default Φ and Γ and no knowledge of the compression/expansion state of the elements in a problem,

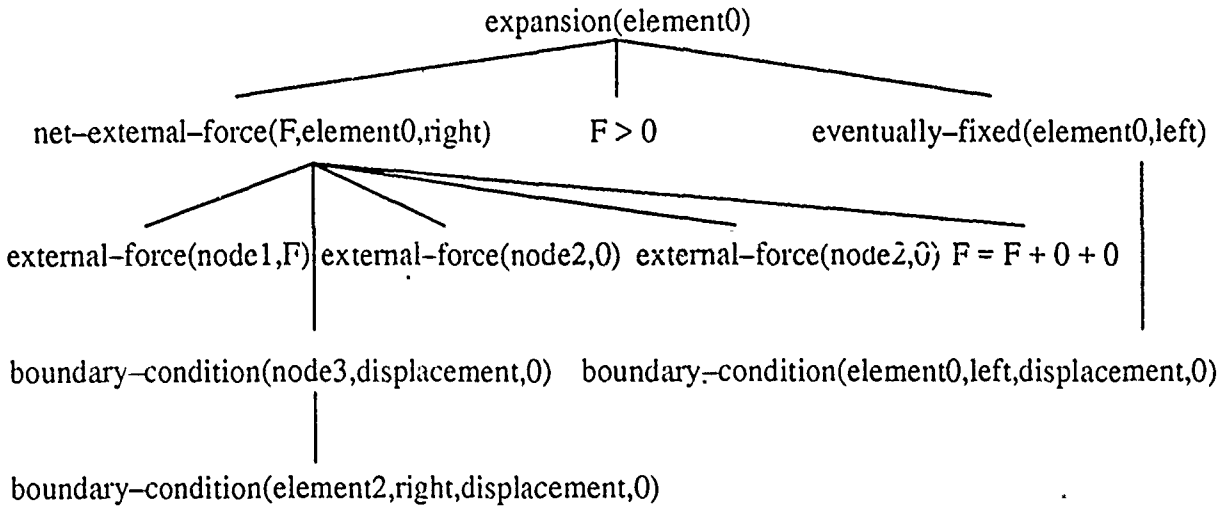


Figure 7. Explanation of Element 0 Being under Expansion

- FEOS–USECE, which is FEOS with Φ and Γ modified as described above and knowledge of the compression/expansion state of the elements in a problem, and
- FEOS–REPORT, which is FEOS with the default Φ and Γ, knowledge of the compression/expansion state of the elements in a problem, and a modification to the main loop so that any time an element is not in its final state the fact is reported to the screen.

It is unclear whether having the compression/expansion information helps FEOS converge faster or not. Figure 8 shows the result of a test of the hypothesis. The bottom axis represents a set of examples with roughly increasing complexity to the right, and the left axis represents the number of messages propagated. The results show no systematic comparative out–performance or under–performance between FEOS and FEOS–USECE in the number of propagations needed to converge on a solution.

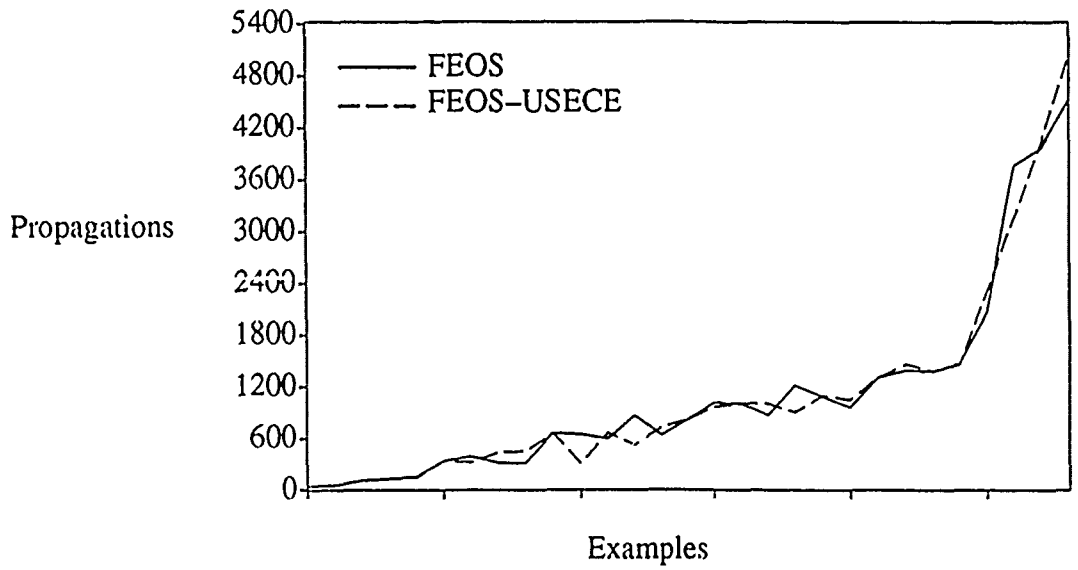


Figure 8. Evaluation of Compression/Expansion State Knowledge on General Examples

The low, and sometimes negative, utility of this information may be due to the fact that the heuristic built into the solver tends to lead elements into the final state and thus little is gained. Conversely, it may be that certain problems require that elements be in states other than their final state during convergence, while others require that the elements be in their final state during convergence. The latter of these types of problems would thus gain by knowing the final states and the former would lose by knowing this information and converge more slowly. One way to determine which of the two possible explanations given above is the case is to design a set of examples for which solving by FEOS–REPORT yields a large number of out–of–state reports. If the former of the explanations for the low utility of the state information is the case, then the state information would cause the convergence for these examples to be improved. One such set of examples is that comprised of examples of the form shown in Figure 9. For these examples, the elements between the left and right

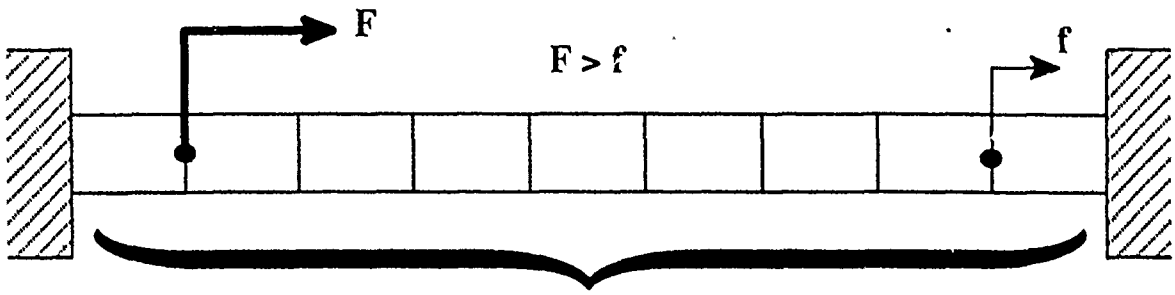


Figure 9. The Form of Examples Usually Out of State

forces will eventually be under compression because the left force is greater than the right force. However, those elements just to the left of the right force will receive messages which cause them to be in the expansion state when the problem is being solved without the state information. When FEOS solves this example set using the state information, there is a degradation in convergence performance, shown in Figure 10. This suggests that the later explanation for the low utility of the state information is the case.

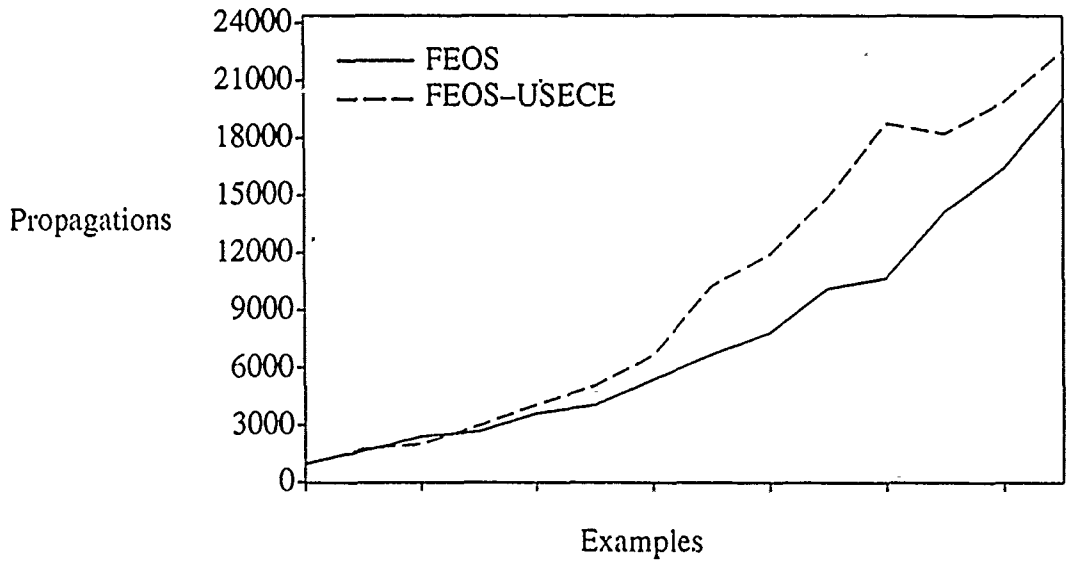


Figure 10. Evaluation Of Compression/Expansion State Knowledge On Examples Usually Out of State

5. PREMESHING

Many of the approaches discussed in Section 2.4 on meshing issues are as yet unexplored. However, one meshing method which has been investigated involves qualitatively preprocessing an initial mesh to determine boundaries that are removable by the replacement of bordering elements with a quantitatively identical element. This method assumes that some nodes are not removable because the parameter values at these nodes are goals of the analysis. Therefore, the method is given goals in terms of nodes that are not removable. This method examines each boundary to determine whether or not it is removable given the goals, a specification for the problem, and a boundary removal domain theory for the problem class. If the boundary is removable, then the elements that share that boundary are replaced with a single element across the boundary. For example, consider Example 3.1 from Logan [Logan86] shown in Figure 11, and the boundary removal domain theory

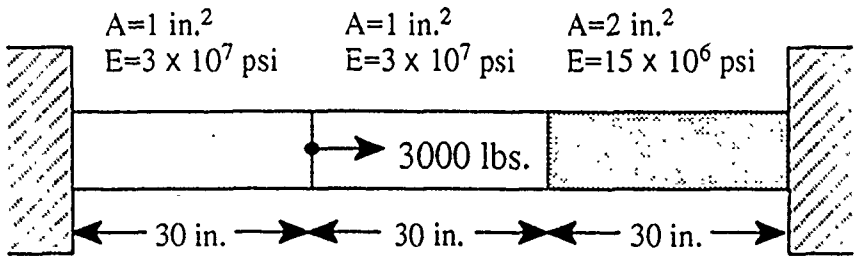


Figure 11. Example 3.1 from Logan [Logan86] of the Example Problem Class

shown in Table 5. The boundary at node 2 can be removed and the problem reduced to a two-element problem by replacing elements 1 and 2 with a single element with aggregate length, length-averaged cross-sectional area, and volume-averaged modulus of elasticity.

Table 5. A Boundary Removal Domain Theory

greater(Node,0) ∧ number–of–nodes(NumberOfElements) ∧
less(Node,NumberOfElements) ∧ no–goals(Node) ∧ no–discontinuities(Node)
⇒ removable–boundary(Node)
goals(Goals) ∧ not–member(Node,Goals) ⇒ no–goals(Node)
external–forces(Node,0) ∧ boundary–condition(Node,force)
⇒ no–discontinuities(Node)

This method is implemented and results in obvious improvement in terms of propagations, shown in Figure 12, because it reduces the problem to one containing fewer elements. The bottom

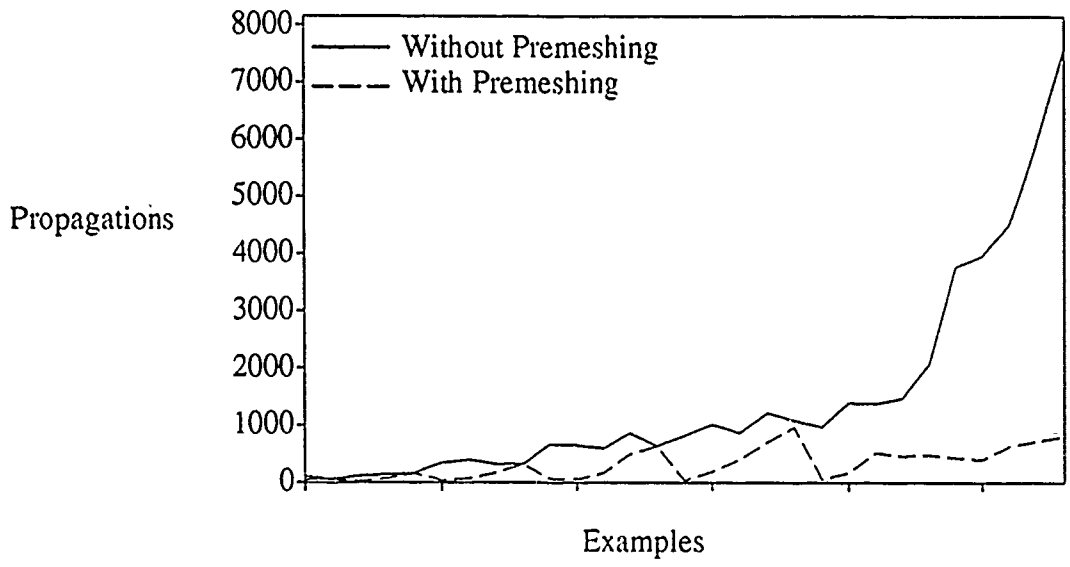


Figure 12. Evaluation of Premeshing by Propagations

axis is an example set with roughly increasing complexity to the right. Since time is consumed in preprocessing the mesh, simply reducing the number of propagations does not constitute a net

improvement. Figure 13 shows the CPU time for the same set of problems with and without the re-

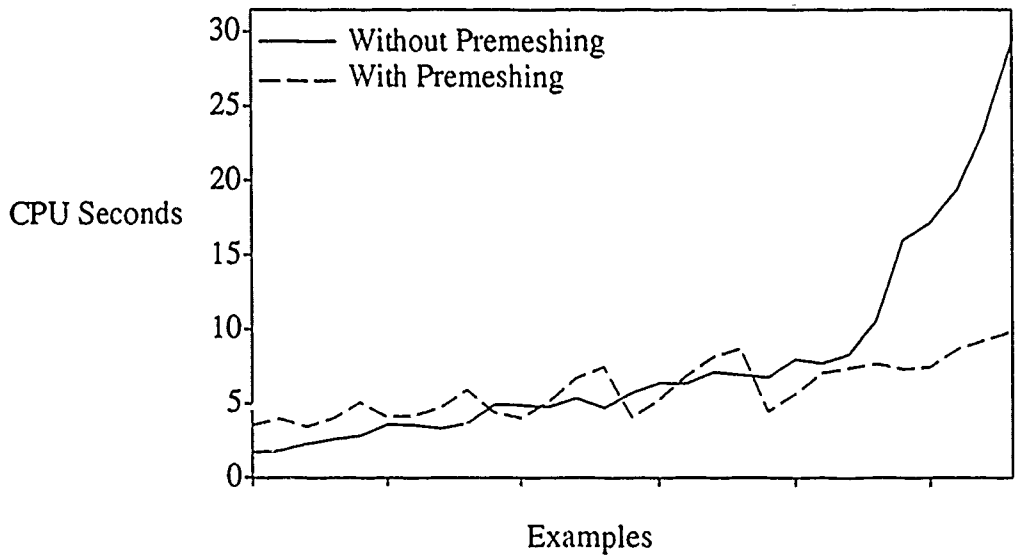


Figure 13. Evaluation Of Premeshing By CPU Time

meshing. As can be seen, although the premeshing times are comparable to the nonpremeshing times for easier problems, the asymptotic effect is a significant net improvement.

6. CONCLUSIONS AND FUTURE WORK

FEOS is a novel finite element solver that appears to be a good vehicle for the application of AI to finite element analysis. The use of qualitative domain theories for learning constraints on elements seems to be a reasonable approach to constraining underconstrained decisions at the elemental level. Aside from intelligent premeshing, no clear path to improvement of finite element analysis programs has been found. However, the space of possibilities has been mapped out and experimentation into the different possibilities is the true test of the feasibility of the task.

In a somewhat related work, Abelson et al. [Abelson89] have shown that AI techniques can be used to aid scientists in scientific computing by developing tools that are useful when setting up and running numerical experiments. They do not, however, address finite element analysis directly.

Dynamic meshing, as well as many other meshing options, has not been studied in depth. This area would seem to have a good chance of having justifiable decisions because scientists must explicitly reason about what does and does not make a good mesh. The design of a domain theory for these decisions and operations in FEOS that can use this theory to fuse and split elements is an obvious path for future research.

The utility of qualitative states is not well understood. Careful study of the examples for which the compression/expansion state information is useful and examples for which it is harmful is necessary. Common aspects in the two sets might be insights into what these states are actually contributing. Additionally, other qualitative states might be identifiable and usable.

All parameter changes in an element are directly explainable in terms of the message just propagated to it. Therefore, if all parameter changes are explained, a large explanation can be

constructed that has these individual parameter change explanations as subexplanations. It may be possible to recognize loops or repetitions comprised of multiple parameter changes that can be reduced to a single parameter change. To do this, these loops must be explicable in terms of eventual convergence, and detectable by an element in a convergence step. If a domain theory that can do this can be written, this approach may be extremely useful. One possibility is the use of axioms similar to those used by Minton et al. [Minton89] in the planning domain. These axioms might codify the nonutility of iterations that move away from the eventual correct answer and the utility of moving toward the eventual solution.

In the equation derivation approach of Chapter 3 the equations became too complex for efficient development and usage in realistically complex problems. The operationality criterion used is that all expressions must be in terms of elemental parameters and that all preconditions be in terms of single elements. One way to attempt to reduce the size of the rules is to use a more general operationality criterion in an attempt to generate rules that involve relatively few calculations and preconditions. This generality in the operationality criterion may be obtained by allowing statements about groups of elements. The exponential growth of the rule size is then in the number of discernable groups of elements instead of elements. This may be a better measure of the true complexity of a problem. Another approach to reducing the complexity of these rules is using assumptions. If certain assumptions are made, certain subexpressions and preconditions may be eliminated by approximation and assumption, respectively. Since these rules are approximate, they can be used to produce "ball park" numbers to move toward whenever a compensation decision is encountered and the two values that comprise the incompatibility are sufficiently far apart.

Section 2.2 identified several decisions that FEOS faces in its convergence procedure. An untried approach is the attempt to learn directly for these decisions instead of learning constraints that constrain the choices embodied by these decisions. However, a domain theory that codifies justifications for these decisions has not been developed to date, and in fact the task appears to be very difficult. In addition, given a problem that eventually converged to a solution, it is unclear which iterations can be considered positive examples to be learned from

An interesting alternative brought up by the use of FEOS involves meshes which are nonhomogeneous in terms of local constraints. If an initial approximation yields a simplified set of local constraints that eventually is very inaccurate for a subset of the elements in a mesh, this set of local constraints can be replaced with a more complex set for the affected elements. In this way, efficiency is accomplished by having the more complex set of constraints for only those elements that require them and by having the elemental parameters determined by the simplified constraints until the inaccuracy builds up to the point where the more complex constraints are necessary. Additionally, the decision to switch to and from the more complex constraints can be treated as a learning task, thus allowing improvement over time.

Most of the techniques reported in this thesis attempt to identify decision points in FEOS and how a system could reason about making these decisions intelligently. Another possibility is the construction of a qualitative domain theory in terms of qualitative proportionalities to determine what conclusions, besides the identified qualitative states, can be made.

A last area for future analysis is in parallelization of FEOS. Since it is object-oriented, massive small-grain parallelism may be effectively exploited. Since elements communicate only with

neighbors, an N–cube configuration with an element at each node is natural to implement and much cheaper, in terms of communication costs, than other algorithms which may require a cross–bar switch. Additionally, geometric reasoning could group the elements so that minimal communication will occur between the groups. Such groups are good candidates for large–grain parallelism.

APPENDIX A. FEOS CODE

The FEOS code is implemented in Lucid Common Lisp on an IBM RT. The elemental object definition is

```
(defflavor element
  (number
   modulus
   area
   length
   (left-element nil)
   (right-element nil)
   (internal-force 0)              ;positive for expansion
   (left-external-force 0)
   (right-external-force 0)
   (left-displacement 0)
   (right-displacement 0)
   (ld-boundary-condition nil)
   (rd-boundary-condition nil))
  ()
  :settable-instance-variables)    ;automatically generates
                                   ;methods for var query,
                                   ;setting, & init
```

The following is the FEOS code. Sample inputs follow.

```
(defvar *propagations* 0)
(defvar *no-changes* t)

(defvar *net-force* 0)             ;constrains rigid body motion
(defvar *nf-counter* 0)
(defvar *nf-file* nil)

(defmethod (element :apply-left-force) (amount update-nf)
  (if ld-boundary-condition
   (print (list number 'error0-apply-left-force)))
  (cond (update-nf
   (incf *net-force* (- amount left-external-force))
   (format *nf-file* "~s ~s~%"
    (incf *nf-counter*) *net-force*)))
  (decf internal-force (/ (- amount left-external-force)
                          (if left-element 2 1)))
  (setq left-displacement
```

```
    (- right-displacement (/ (* length internal-force)
                             area modulus)))
    (setq left-external-force amount)
    (cond
     ((not right-element)
      (incf *net-force* (- internal-force right-external-force))
      (format *nf-file* "~s ·s~%"
       (incf *nf-counter*) *net-force*)
      (setq right-external-force internal-force))))
  (defmethod (element :apply-right-force) (amount update-nf)
    (if rd-boundary-condition
     (print (list number 'error0-apply-right-force)))
    (cond (update-nf
     (incf *net-force* (- amount right-external-force))
     (format *nf-file* "~s ~s~%"
      (incf *nf-counter*) *net-force*)))
    (incf internal-force (/ (- amount right-external-force)
                            (if right-element 2 1)))
    (setq right-displacement
     (+ left-displacement (/ (* length internal-force)
                             area modulus)))
    (setq right-external-force amount)
    (cond
     ((not left-element)
      (incf *net-force* (- (+ internal-force
                              left-external-force)))
      (format *nf-file* "~s ~s~%"
       (incf *nf-counter*) *net-force*)
      (setq left-external-force (- internal-force)))))
  (defmethod (element :left-guesses) (force displacement)
    (let* ((k (/ (* area modulus) length))
      (displacement-delta (- displacement left-displacement))
      (force-delta (- (- force internal-force)
                      left-external-force))
      (d-disparity (> (abs displacement-delta) d-threshold))
      (f-disparity (> (abs force-delta) f-threshold))
      discrepancy)
     (cond
      ((and ld-boundary-condition d-disparity)
       (print (list number 'error0-left-guesses)))
      ((and ld-boundary-condition f-disparity)
       (send self :inc-lef force-delta)
       (setq *no-changes* nil))
```

```
((and d-disparity f-disparity)
 (let ((delta (/ (- force-delta (* k displacement-delta))
                 2)))
      (incf internal-force delta)
      (if rd-boundary-condition
        (send self :inc-ref delta))
      (decf left-displacement (/ delta k)))
 (setq *no-changes* nil))
((and d-disparity rd-boundary-condition)
 (incf left-displacement (/ displacement-delta 2))
 (decf internal-force (/ (* k displacement-delta) 2))
 (send self :inc-ref (- (/ (* k displacement-delta) 2)))
 (setq *no-changes* nil))
(d-disparity
 (incf left-displacement (/ displacement-delta 2))
 (decf right-displacement (/ displacement-delta 4))
 (decf internal-force (/ (* k displacement-delta) 4))
 (setq *no-changes* nil))
((and f-disparity rd-boundary-condition)
 (incf internal-force (/ force-delta 2))
 (send self :inc-ref (/ force-delta 2))
 (decf left-displacement (/ force-delta (* 2 k)))
 (setq *no-changes* nil))
(f-disparity
 (incf internal-force (/ force-delta 2))
 (decf left-displacement (/ force-delta (* 4 k)))
 (incf right-displacement (/ force-delta (* 4 k)))
 (setq *no-changes* nil))
((> (abs (setq discrepancy
     (- internal-force
        (* k (- right-displacement left-displacement)))))
   f-threshold)
 (decf internal-force discrepancy)
 (if ld-boundary-condition
   (send self :inc-lef (- discrepancy)))
 (if rd-boundary-condition
  (send self :inc-ref discrepancy))))))

(defmethod (element :right-guesses) (force displacement)
  (let* ((k (/ (* area modulus) length))
    (displacement-delta (- displacement right-displacement))
    (force-delta (- (- internal-force force)
                    right-external-force))
    (d-disparity (> (abs displacement-delta) d-threshold))
```

```
 (f-disparity (> (abs force-delta) f-threshold))
 discrepancy)
(cond
 ((and d-disparity rd-boundary-condition)
  (print (list number 'error0-right-guesses)))
 ((and f-disparity rd-boundary-condition)
  (send self :inc-ref force-delta)
  (setq *no-changes* nil))
 ((and d-disparity f-disparity)
  (let ((delta (/ (- (* k displacement-delta) force-delta)
                  2)))
       (incf internal-force delta)
       (if ld-boundary-condition
        (send self :inc-lef (- delta)))
       (incf right-displacement (/ delta k)))
  (setq *no-changes* nil))
 ((and d-disparity ld-boundary-condition)
  (incf right-displacement (/ displacement-delta 2))
  (incf internal-force (/ (* k displacement-delta) 2))
  (send self :inc-lef (- (/ (* k displacement-delta) 2)))
  (setq *no-changes* nil))
 (d-disparity
  (incf right-displacement (/ displacement-delta 2))
  (decf left-displacement (/ displacement-delta 4))
  (incf internal-force (/ (* k displacement-delta) 4))
  (setq *no-changes* nil))
 ((and f-disparity ld-boundary-condition)
  (decf internal-force (/ force-delta 2))
  (send self :inc-lef (/ force-delta 2))
  (decf right-displacement (/ force-delta (* 2 k)))
  (setq *no-changes* nil))
 (f-disparity
  (decf internal-force (/ force-delta 2))
  (decf right-displacement (/ force-delta (* 4 k)))
  (incf left-displacement (/ force-delta (* 4 k)))
  (setq *no-changes* nil))
 ((> (abs (setq discrepancy
      (- internal-force
         (* k (- right-displacement left-displacement)))))
    f-threshold)
  (decf internal-force discrepancy)
  (if rd-boundary-condition
    (send self :inc-ref discrepancy))
```

```
      (if ld-boundary-condition
        (send self :inc-lef (- discrepancy))))))))

(defmethod (element :inc-lef) (amount)
  (incf left-external-force amount)
  (incf *net-force* amount)
  (format *nf-file* "~s ~s~%" (incf *nf-counter*) *net-force*)
  (if left-element
      (send left-element :set-right-external-force
                         left-external-force)))

(defmethod (element :inc-ref) (amount)
  (incf right-external-force amount)
  (incf *net-force* amount)
  (format *nf-file* "~s ~s~%" (incf *nf-counter*) *net-force*)
  (if right-element
      (send right-element :set-left-external-force
                          right-external-force)))

(defmethod (element :propagate-left) ()
  (cond
   (left-element
    (incf *propagations*)
    (send left-element :right-guesses
          internal-force left-displacement))))

(defmethod (element :propagate-right) ()
  (cond
   (right-element
    (incf *propagations*)
    (send right-element :left-guesses
          internal-force right-displacement))))

(defmethod (element :output) ()
  (format t "~%Element ~s:~%external-
forces: left: ~s right: ~s~%internal-force: ~s~%displace-
ments: left: ~s right: ~s~%"
     number left-external-force right-external-force
     internal-force left-displacement right-displacement))

(defmethod (element :satisfies-constraints) ()
  (< (- internal-force
        (* (/ (* area modulus) length)
           (- right-displacement left-displacement)))
     f-threshold))

(defun calculate (elements)
  (let (mesh
```

```
  (iterations 0)
  (start-time (get-internal-run-time)))
      (setq *propagations* 0)
      (if *nf-file* (close *nf-file*))
      (setf *nf-file* (open "nf"
             :direction :output
             :if-exists :rename))
      (setf *nf-counter* 0)
      (setq *net-force* 0)
      (format *nf-file* "~s ~s~%"
              (incf *nf-counter*) *net-force*)
      (if (satisfies-constraints
           (setq mesh (generate-mesh (rest elements)))
         (if (satisfies-constraints
         (output-mesh
          (progn
           (format t "~%Mesh initially stable")
           (format t "~%Calculating ~s element mesh"
                     (list-length mesh))
           (setq *no-changes* nil)
           (loop (incf iterations)
            (if (= (truncate (/ iterations 5)) (/ iterations 5))
                (format t "."))
            (setq *no-changes* t)
            (dolist (element mesh)
               (send element :propagate-left)
               (send element :propagate-right))
            (if *no-changes* (return mesh))))))
             (format t "~%Solution satisfies constraints")
             (format t
              "~%Solution does not satisfy constraints"))
         (format t "~%Mesh initially unstable"))
          (format t "~%Net force: ~s" *net-force*)
          (format t "~%Elaspsed time: ~s"
                  (time-since start-time))
          (format t "~%Propagations: ~s" *propagations*)
          (format t "~%Iterations: ~s" iterations)
          (close *nf-file*)))

(defun apply-external-forces (mesh elements)
  (let (force (node 0))
       (dolist (element elements mesh)
          (if (setq force (first element))
            (progn
```

```
          (format t "~%Applying a force of ~s to node ~s"
                  force node)
          (send (nth node mesh) :apply-left-force force t)
          (if (> node 0)
              (send (nth (1- node) mesh)
                    :apply-right-force force nil))))
          (if (setq force (seventh element))
          (progn
          (format t "~%Applying a force of ~s to node ~s"
                 force (1+ node))
          (send (nth (1+ node) mesh)
                :apply-right-force force t)
          (if (\= node (list-length mesh))
              (send (nth (+ node 2) mesh)
                    :apply-left-force force nil))))
          (incf node))))

(defun time-since (tm) ; how many seconds since tm was set?
  (float (/ (- (get-internal-run-time) tm)
            internal-time-units-per-second)) )

(defun generate-mesh (elements)
  (apply-external-forces
   (link-elements (make-elements elements)) elements))

(defun make-elements (elements)
  (let ((head (first elements))
   (tail (rest elements)))
       (cons (make-instance 'element
             :ld-boundary-condition (second head)
             :area (third head)
             :modulus (fourth head)
             :length (fifth head)
             :rd-boundary-condition (sixth head))
        (if tail (make-elements tail)))))

(defun link-elements (mesh)
  (let* ((head (first mesh))
    (neck (second mesh))
    (neck-tail (rest mesh))
    (tail (rest neck-tail))
    hair)
   (send head :set-right-element neck)
   (send neck :set-left-element head)
   (send head :set-rd-boundary-condition
         (send neck :ld-boundary-condition))
```

```
    (if (setq hair (send head :left-element))
        (send head :set-number (1+ (send hair :number)))
        (send head :set-number 0))
    (if tail
        (link-elements neck-tail)
        (send neck :set-number (1+ (send head :number))))
    mesh))

(defun output-mesh (mesh)
  (let ((tail (rest mesh)))
       (send (first mesh) :output)
       (if tail (output-mesh tail))
       mesh))

(defun satisfies-constraints (mesh)
  (let ((result t))
       (dolist (element mesh result)
          (if (not (send element :satisfies-constraints))
            (setq result nil)))))
```

The following is a sample input Example 3.1 from Logan [Logan86].

```
(defconstant d-threshold 0.00001)
(defconstant f-threshold 1)

; Problems are input as a list whose first member is a list
; of the compression/expansion states of the problem elements
; and the rest of the list is elements.  Each element is a
; list of its features in the following order:
; left-external-force ld-boundary-condition
; area modulus length
; rd-boundary-condition right-external-force
; Any properties not needed at the end are accepted as nil.

(defun eg3.1 ()
  '((e c c r l l)(nil t 1 30E6 30)
    (3000 nil 1 30E6 30)
    (nil nil 2 15E6 30 t)))
```

APPENDIX B. EQUATION DERIVATION DOMAIN THEORY

The format of the following rules is

```
(<= consequent
  (antecedents))
```

where `consequent` and `antecedents` are predicates of the form

```
(predicate arguments)
```

and all variables begin with a question mark (?). The domain theory rules follow, and the facts pursuant to an example follow the rules.

```
(<= (ratio ?e ?r)
  ((left-boundary-condition ?e force)
   (left-displacement ?e ?ld)
   (right-boundary-condition ?e force)
   (right-displacement ?e ?rd)
   (= ?r (/ ?ld ?rd))))
(<= (left-displacement ?e ?ld)
  ((= ?x (1- ?e))
   (left-fixed ?le ?x)
   (effective-k ?le ?x ?lk)
   (must-absorb ?le ?x ?lf)
   (= ?ld (- (/ ?lf ?lk)))))
(<= (right-displacement ?e ?rd)
  ((= ?x (1+ ?e))
   (right-fixed ?x ?re)
   (effective-k ?x ?re ?rk)
   (must-absorb ?x ?re ?rf)
   (= ?rd (- (/ ?rf ?rk)))))
(<= (left-external-force ?e ?f)
  ((left-boundary-condition ?e displacement)
   (right-boundary-condition ?e force)
   (must-absorb ?e ?f)))
```

```
(<= (left-fixed ?x ?y)
  ((left-boundary-condition ?x displacement)
   (less-or-equal ?x ?y)
   (lf-clear ?x ?y)))

(<= (lf-clear ?x ?y)
  ((= ?x ?y)
   (right-boundary-condition ?x force)))

(<= (lf-clear ?x ?y)
  ((right-boundary-condition ?x force)
   (= ?z (1+ ?x))
   (lf-clear ?z ?y)))

(<= (right-fixed ?x ?y)
  ((right-boundary-condition ?y displacement)
   (less-or-equal ?x ?y)
   (rf-clear ?x ?y)))

(<= (rf-clear ?x ?y)
  ((= ?x ?y)
   (left-boundary-condition ?x force)))

(<= (rf-clear ?x ?y)
  ((left-boundary-condition ?y force)
   (= ?z (1- ?y))
   (rf-clear ?x ?z)))

(<= (must-absorb ?x ?y ?f)
  ((= ?x ?y)
   (must-absorb ?x ?f)))

(<= (must-absorb ?x ?y ?f)
  ((= ?z (1+ ?x))
   (must-absorb ?x ?f1)
   (must-absorb ?z ?y ?f2)
   (= ?f (+ ?f1 ?f2))))

(<= (must-absorb ?e ?f)
  ((all-external-forces ?af)
   (aggregate-absorption ?e ?af ?f)))

(<= (aggregate-absorption ?e ((?nx ?fx)) ?f)
  ((absorption ?e ?nx ?fx ?f)))

(<= (aggregate-absorption ?e ((?nx ?fx) . ?fr) ?f)
  ((aggregate-absorption ?e ?fr ?f1)
   (absorption ?e ?nx ?fx ?f2)
   (= ?f (+ ?f1 ?f2))))
```

```
(<= (absorption ?e ?n ?ef 0)
  ((less ?e ?n) ;?e left of ?n
   (= ?x (1- ?n))
   (left-fixed ?le ?x)
   (less ?e ?le)))

(<= (absorption ?e ?n ?ef ?f)
  ((less ?e ?n) ;left
   (= ?x (1- ?n))
   (left-fixed ?le ?x)
   (less-or-equal ?le ?e)
   (side-share ?le ?x ?n ?ef ?sf)
   (element-share ?e ?le ?x ?sf ?f)))

(<= (absorption ?e ?n ?ef 0)
  ((less-or-equal ?n ?e) ;right
   (right-fixed ?n ?re)
   (less ?re ?e)))

(<= (absorption ?e ?n ?ef ?f)
  ((less-or-equal ?n ?e) ;right
   (right-fixed ?n ?re)
   (less-or-equal ?e ?re)
   (side-share ?n ?re ?n ?ef ?sf)
   (element-share ?e ?n ?re ?sf ?f)))

(<= (side-share ?le ?x ?n ?ef ?sh)
  ((less ?le ?n) ;left
   (effective-k ?le ?x ?lk)
   (right-fixed ?n ?re)
   (effective-k ?n ?re ?rk)
   (= ?sh (* (/ ?lk (+ ?lk ?rk)) (- ?ef)))))

(<= (side-share ?y ?re ?n ?ef ?sh) ;right
  ((= ?y ?n)
   (effective-k ?y ?re ?rk)
   (= ?z (1- ?y))
   (left-fixed ?x ?z)
   (effective-k ?x ?z ?lk)
   (= ?sh (* (/ ?lk (+ ?lk ?rk)) (- ?ef)))))

(<= (effective-k ?x ?y ?k)
  ((= ?x ?y)
   (value ?x k ?k)))

(<= (effective-k ?x ?y ?k)
  ((e-k ?x ?y ?l ?k)))
```

```
(<= (e-k ?x ?y ?l ?k)
  ((= ?x ?y)
   (value ?x k ?k)
   (value ?x l ?l)))

(<= (e-k ?x ?y ?l ?k)
  ((value ?x k ?k1)
   (value ?x l ?l1)
   (= ?re (1+ ?x))
   (e-k ?re ?y ?lr ?kr)
   (= ?l (+ ?l1 ?lr))
   (= ?k (/ (+ (* ?l1 ?l1 ?k1) (* ?lr ?lr ?kr))
   (* (+ ?l1 ?lr) (+ ?l1 ?lr))))))

(<= (element-share ?e ?x ?y ?tf ?f)
  ((= ?e ?x)
   (= ?e ?y)
   (= ?f ?tf)))

(<= (element-share ?e ?x ?y ?tf ?f)
  ((= ?e ?x)
   (= ?z (1+ ?e))
   (effective-k ?z ?y ?rk)
   (value ?e k ?ek)
   (= ?f (* (/ ?ek (+ ?ek ?rk)) ?tf))))

(<= (element-share ?e ?x ?y ?tf ?f)
  ((= ?e ?y)
   (= ?z (1- ?e))
   (effective-k ?x ?z ?lk)
   (value ?e k ?ek)
   (= ?f (* (/ ?ek (+ ?ek ?lk)) ?tf))))

(<= (element-share ?e ?x ?y ?tf ?f)
  ((= ?z1 (1- ?e))
   (= ?z2 (1+ ?e))
   (effective-k ?x ?z1 ?lk)
   (effective-k ?z2 ?y ?rk)
   (value ?e k ?ek)
   (= ?f (* (/ ?ek (+ ?lk ?ek ?rk)) ?tf))))
```

The following is a set of facts that describe example 3.1 from Logan [Logan86] shown in Figure 11.

```
(all-external-forces ((1 3000.0)))
(number-of-elements 3)
```

```
(value 0 k 1.0e6)
(value 0 l 30.0)
(left-boundary-condition 0 displacement)
(right-boundary-condition 0 force)
(value 1 k 1.0e6)
(value 1 l 30.0)
(left-boundary-condition 1 force)
(right-boundary-condition 1 force)
(value 2 k 1.0e6)
(value 2 l 30.0)
(left-boundary-condition 2 force)
(right-boundary-condition 2 displacement)
```

APPENDIX C. QUALITATIVE STATE DOMAIN THEORY

The format of the following rules is the same as those in APPENDIX B.

```
(<= (ce-state ?e expansion)
  ((left-fixed ?le ?e)
   (net-force ?le ?e 0)
   (right-fixed ?e ?re)
   (net-force ?e ?re ?fr)
   (< 0 ?fr)))

(<= (ce-state ?e compression)
  ((left-fixed ?le ?e)
   (net-force ?le ?e 0)
   (right-fixed ?e ?re)
   (net-force ?e ?re ?fr)
   (< ?fr 0)))

(<= (ce-state ?e expansion)
  ((right-fixed ?e ?re)
   (net-force ?e ?re 0)
   (left-fixed ?le ?e)
   (net-force ?le ?e ?fl)
   (< ?fl 0)))

(<= (ce-state ?e compression)
  ((right-fixed ?e ?re)
   (net-force ?e ?re 0)
   (left-fixed ?le ?e)
   (net-force ?le ?e ?fl)
   (< 0 ?fl)))

(<= (left-fixed ?x ?y)
    ((left-boundary-condition ?x displacement)
     (<= ?x ?y)
     (lf-clear ?x ?y)))

(<= (lf-clear ?x ?y)
    ((= ?x ?y)))

(<= (lf-clear ?x ?y)
    ((right-boundary-condition ?x force)
     (= ?z (1+ ?x))
     (lf-clear ?z ?y)))
```

```
(<= (right-fixed ?x ?y)
    ((right-boundary-condition ?y displacement)
     (<= ?x ?y)
     (rf-clear ?x ?y)))

(<= (rf-clear ?x ?y)
    ((= ?x ?y)))

(<= (rf-clear ?x ?y)
    ((left-boundary-condition ?y force)
     (= ?z (1- ?y))
     (rf-clear ?x ?z)))

(<= (net-force ?x ?y ?f)
  ((all-external-forces ?fs)
   (nf ?x ?y ?fs ?f)))

(<= (nf ?x ?y ((?n ?f)) 0)
  ((<= ?n ?x)))

(<= (nf ?x ?y ((?n ?f)) 0)
  ((< ?y ?n)))

(<= (nf ?x ?y ((?n ?f)) ?f)
  ((< ?x ?n)
   (<= ?n ?y)))

(<= (nf ?x ?y ((?n1 ?f1). . ?fs) ?f)
  ((< ?x ?n1)
   (<= ?n1 ?y)
   (nf ?x ?y ?fs ?f2)
   (= ?f (+ ?f1 ?f2))))

(<= (nf ?x ?y ((?n1 ?f1) . ?fs) ?f)
  ((<= ?n1 ?x)
   (nf ?x ?y ?fs ?f)))

(<= (nf ?x ?y ((?n1 ?f1) . ?fs) ?f)
  ((< ?y ?n1)
   (nf ?x ?y ?fs ?f)))
```

REFERENCES

[Abelson89] H. Abelson, M. Eisenberg, M. Halfant, J. Katzenelson, E. Sacks, G. Sussman, J. Wisdom and K. Yip, "Intelligence in Scientific Computing," *Communications of the Association for Computing Machinery 32*, (May 1989), pp. 546–562.

[DeJong86] G. F. DeJong and R. J. Mooney, "Explanation–Based Learning: An Alternative View," *Machine Learning 1*, 2 (April 1986), pp. 145–176. (Also appears as Technical Report UILU–ENG–86–2208, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana–Champaign.)

[Forbus84] K. D. Forbus, "Qualitative Process Theory," *Artificial Intelligence 24*, (1984), pp. 85–168.

[LaLonde90] W. R. LaLonde and J. R. Pugh, *Inside Smalltalk*, Prentice Hall, 1990.

[Logan86] D. L. Logan, *A First Course in the Finite Element Method*, PWS–Kent, Boston, 1986.

[Minton89] S. Minton, J. G. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni and Y. Gil, "Explanation–Based Learning: A Problem Solving Perspective," *Artificial Intelligence 40*, (1989), pp. 63–118.

[Mitchell86] T. M. Mitchell, R. Keller and S. Kedar–Cabelli, "Explanation–Based Generalization: A Unifying View," *Machine Learning 1*, 1 (January 1986), pp. 47–80.

[Mooney86] R. J. Mooney and S. W. Bennett, "A Domain Independent Explanation–Based Generalizer," *Proceedings of the National Conference on Artificial Intelligence*, Philadelphia, PA, August 1986, pp. 551–555. (Also appears as Technical Report UILU–ENG–86–2216, AI Research Group, Coordinated Science Laboratory, University of Illinois at Urbana–Champaign.)

[Simmons86] R. Simmons, "'Commonsense' Arithmetic Reasoning," *Proceedings of the National Conference on Artificial Intelligence*, Philadelphia, PA, August 1986, pp. 118–124.